DOCUMENTATION MANAGEMENT AND GENERATION FOR DOMAIN-SPECIFIC MODELS

By

Kiran Kumar Guragain

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Computer Science

August, 2010

Nashville, Tennessee

Approved:

Dr. Akos Ledeczi

Dr. Gabor Karsai

DEDICATION

Dedicated to my beloved parents Bharat Prasad Guragain and Sunita Guragain

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Akos Ledeczi, for providing guidance and support during my graduate education at Vanderbilt University. I am grateful to, Dr. Gabor Karsai, for his insightful comments and suggestions on my Master's Thesis. I appreciate all the help and support that I received from my friends, fellow graduates and staff members at Institute for Software Integrated Systems.

I am thankful to my mother, Sunita Guragain, and father, Bharat Prasad Guragain, for their inspiration and encouragement during my graduate studies.

TABLE OF CONTENTS

DEDICATION I

ACKNOWLEDGEMENTS II

TABLE OF CONTENTS III

LIST OF FIGURES V

I. INTRODUCTION 1

II. RELATED WORK 4

Hyperdoc 5

Javadoc 5

Simulink Web View 6

Microsoft Visio 7

IBM Rational 7

III. BACKGROUND 9

Design and Architecture 10

IV. IMPLEMENTATION 14

Interpreter 14

HTML/ JavaScript 14

Model Documenter 15

Advantages of using HTML 16

Recording Documentation/Comments 16

External Editor 18

Syncing Source and Design View 19

Web Exporter 20

Exporting model image 21
Conversion to PNG 21
Exporting Model Locations 23
Mapping location 24
Exporting to HTML 25
Generating the Tree Browser 26
Navigation 27
V. CASE STUDY 29
Signal Flow Example 29
VI. SUMMARY AND FUTURE WORK 33
REFERENCES 34

LIST OF FIGURES

Figure 1 General System Architecture 11

Figure 2 Model Documenter Architecture 12

Figure 3 Web Exporter Architecture 13

Figure 4 Model Documenter Window 16

Figure 5 Example of saving documentation in registry 18

Figure 6 Original model as it appears in GME 22

Figure 7 Exported model from GME 23

Figure 8 Object Hierarchy and corresponding HTML code which is read by Yahoo library to construct tree browser 27

Figure 9 Model Documenter window that takes documentation 29

Figure 10 Dialog asking to select a folder 30

Figure 11 Exported HTML in Web Browser 31

CHAPTER I

INTRODUCTION

Documentation is essential to every system as it helps in disseminating knowledge about the system. Automatic Documentation exploits existing design or data, with a little or no effort from the user to create documentation. Automatic documentation has been of interest mainly because of its ability to reduce effort that is required to create documentation. Most of the system designs have enough information that can be exploited to generate some of the documentation. This approach has several advantages such as reduced cost, consistency with the design and ease of maintenance. Though automation might not eliminate cost altogether, it is likely to reduce cost associated with documentation. Cost can still be incurred in areas such as document editing and proof reading because machine generated documents might not be of the highest quality. When a system is designed and documented, it is highly likely that different people might be involved in these jobs; this might result in discrepancies between the design and the documentation. Generating documentation using an existing knowledgebase such as a design itself, helps eliminate this problem. Another pronounced advantage is maintenance; a change in the design simply requires rerunning the automatic documentation generator, assuming that the generated documentation is not modified. This is likely to save a lot of effort needed to update the document and also to prevent inconsistencies that may occur when documents are manually updated.

In case of modeling, documentation can be useful to explain the purpose of the model and the relationships among different parts of the model. The designer can always add information that he thinks is necessary to understand or use the model. This information can act as a means through which the developer of the model communicates with the other designers or users. This thesis describes an automatic documentation generation tool for the Generic Modeling Environment (GME) [12], a domain specific modeling tool developed by Institute of Software Integrated Systems at Vanderbilt University. GME lacks proper documentation generation facility that is relatively common in modeling tools. In this work we have developed one such tool that helps the user describe the modeling objects and generate the corresponding documentation automatically.

The tool runs as a GME interpreter and generates HTML documentation that contains all the graphical model views that can be navigated just like in the GME. As such, it visualizes the model and also presents the textual description the modeler provided. The description can include HTML formatted text and hyperlinks that help explain the model or convey other important information. At a high level, our tool consists of two parts. The first part is the documenter which is used for providing textual description for the models and the second part is the exporter which is used to assemble the graphical model views and the attached description into a set of interlinked web pages.

Manually documenting GME models would be a cumbersome process. For example, this would involve exporting every model as an image and then writing corresponding HTML files that describe the model. Of course, all of these processes cannot be automated as the description to associate with an object needs to be provided by the user. The user is responsible for including documentation for each and every object that he thinks requires it.

Other tasks such as exporting the graphical models and generating HTML pages are performed automatically. This approach has a marked advantage over manual documentation because manually writing, appropriately formatting and publishing all the information is monotonous, take a lot of time and are error prone.

The remainder of this thesis is organized in the following manner. Chapter II describes the existing works in automatic documentation generation. Chapter III introduces the background concepts needed to understand this work and describes our system architecture. Chapter IV discusses our implementation in detail and Chapter V presents a case study of the Signal Flow Model. We conclude by highlighting the future work that can be done in Chapter VI.

CHAPTER II

RELATED WORK

There have been efforts in the past to automate the documentation process. Natural Language Generation [1] is a technique that can generate coherent documentation in multiple languages. NLG based system is used in [1] to generate online interactive documentation and paper manuals. The system is capable of modeling the domain and the linguistics and the user has the ability to control the degree of automation in the generation process. The text generated by the system is a series of steps in which each step is described in a single sentence. The domain is described using an Object Oriented model and each entity is represented by a set of all the elementary components. Each of these elementary components contains description of the events, properties and actions. The user can change properties and descriptions through a GUI. The system also allows existing components to be reused in a new design. For example, some of the specifications of a DVD player can be used for designing the model of a TV (like Power buttons and power connectors). From these specifications, Planner [1] is used to generate a plan which is a sequence of steps that transitions the system from the user specified source state to the goal state. As an automatically generated plan may not be optimal, the system provides capability of editing plans. The plans generated can be grouped to compose the document [1].

Hyperdoc

HyperDoc [2] is a system that generates manuals for interactive systems like tape players. The system is based on Finite State Machines. Each state represents the operation that the system performs and state transition occurs when user presses a button. So, for the system to go from one state to another multiple buttons might be used. For example, when a tape is playing, a user may need to press stop button first (transition to stopped playing state) and then press eject to remove the tape from the system (transition to no tape state). Given a source state and a target state, the system can generate instructions for moving from the source state to the target state. It is used for creating answers to *How–To* questions. The system uses a graph internally and finds the shortest path between the source and the target state and then generates a series of moves that are necessary to bring the system to the target state. In a real system, the moves may involve pressing different buttons. This approach can be used to generate documentation automatically. From each state, the steps to reach the other state can be answered by computing all pairs of the shortest paths for the state machine. However, computing all pairs of the shortest paths may take a long time and generate very lengthy documentation. [2] describes a method that can be used to produce minimal documentation by computing a minimum spanning tree for the graph. This approach is interesting and can save a lot of documentation effort and prevent human errors. However, it can be very complicated for non trivial systems and most of the time it is desirable to have control over the way the documentation is created.

Javadoc

Apart from these earlier approaches, there have been significant efforts in creating documentation automatically, especially in the areas of programming languages and

modeling. Most programming languages such as Java and .NET have tools that can collect information from the code (including comments) and create documentation that is useful for programmers. One popular example is Javadoc, a tool that parses source files for specialized comments, classes and methods and generates a navigable HTML documentation. Javadoc requires that the documentation precede the class, field or the member method definition and be enclosed by /**… */ delimiter. It also supports different tags within the comments to distinguish the method parameters, return type etc.

Simulink Web View

Web View is included in the Simulink Report Generator and can be used to export the Simulink and the Stateflow models into interactive web pages. Web view can be used by anyone and doesn't require the Matlab software to view the models. This enables the developed models to be shared among the users without much difficulty. The export feature in the Simulink is similar to our export tool. It supports navigation of the models using a tree browser. Elements of the model can be clicked for navigation and hovering the mouse pointer over an element displays the attributes of the element using a tooltip. One advantage of Simulink Web View is that, it allows selective export of models. By selective we mean that, it has the option of exporting parts of the model hierarchy. The export options include the overall model, the current model in view, current and above, and current and below. One significant difference from our work is that there is no description associated with the exported model. Though Simulink allows users to enter a description for each Simulink block that they use in the model, the web view doesn't export those descriptions. Even if the description was exported, it supports plain text descriptions only and therefore, proper style and formatting cannot be applied.

The exported pages use frames to divide the page into sections as opposed to our approach of using inline frames (*iframe*). The format used for exporting the image is Scalable Vector Graphics (SVG). SVG is an XML based format and uses vector graphics. Using vector graphics, quality of the image is maintained when it is magnified. Individual elements in SVG graphics are selectable and can react to input events. Also, texts in a SVG image are searchable.

Microsoft Visio

Microsoft Visio is a modeling tool that helps visualize and model complex systems. It supports UML diagrams, circuit diagrams, flowchart, ER diagrams (database modeling), general block diagrams, workflow diagrams, organizational chart and many others. It can automatically export these models into web pages; however the exported pages are not interactive. The page consists of a tree browser which can be used to view the model. The diagrams can also be exported into a pdf file or images. Though it allows a model to have a description, it does not get exported to the web pages. Unlike Simulink, which exports model in the SVG format, Visio can export models in various formats such as GIF, JPEG, PNG, SVG and PNG.

IBM Rational

Exporting models is a relatively common feature in the UML modeling tools.IBM Rational is a very popular product that supports UML/SysML modeling. The UML diagrams can be used to generate code in languages such as Java and C++. IBM also has a separate IBM Rational Publishing Engine which is used to create the documentation automatically. The documentation can be created based on a predefined template. The engine supports graphical template editing and is easy to use. The data for the documentation can come

from various sources such as IBM Rational Models, Rational Doors etc. It can also be used to generate documentation from third party sources if the sources are based on XML format. It supports publishing documentation in different formats such as Word, PDF, and HTML.

The early systems were mostly concerned about automatically generating user manuals. They viewed the system as state machine and computed the necessary steps for transitioning the system from one state to another. These series of steps is then described automatically to create documentation. Most of the current programming languages and modeling tools also have automatic documentation built into them. These systems exploit the presence of information in the source code or model to create documentation. Modeling systems such as Simulink and Microsoft Visio, allow users to associate descriptions with a model. But they do not export the description to the generated documents. The documentation is mainly published in the form of web pages or pdf.

CHAPTER III

BACKGROUND

Model Integrated Computing (MIC) [9] places model at the center of the system lifecycle. Models are created while designing a system and are stored in a model database. Models are used to describe the domain, environment and system architecture. A model interpreter [9], is a software written to analyze the models and generate executables or artifacts that compose the system [11]. Hence, an Interpreter defines the relationship between a problem representation and a solution. MIC consists of various tools that help designers construct a system. These tools include Generic Modeling Environment (GME), Universal Data Model Package (UDM), Graph Rewriting and Transformation Language (GReAT), Design Space Exploration Tool (DESERT) and the Open Tool Integration Framework (OTIF) [10].

GME is a graphical modeling tool that is used to create a domain specific model. This involves creating a metamodel for a domain. This metamodel represents the syntax and semantics of the domain such as the relationship between the components and how they can be connected together. Constraints can be applied to these relationships or properties of an object. Besides syntactic and semantic description, the metamodel can also be used to control the way a model can be viewed. Using metamodel, different models can be created for the domain. The model can be transformed by an interpreter to generate program code, configuration files, or data. The output of an interpreter can be either an end product or an input to other systems. For example, an interpreter might convert GME model

into Simulink model that can be fed into Simulink. GME has component based architecture and supports access to model data through various programmatic interfaces [11].

Design and Architecture

We had several alternatives for the design of the documentation system and the model export tool. Our initial idea was to export all the information contained in the model into an XML file and images. The XML file would serve as a data source for the system. We could use server side scripts such as PHP or Java to read the XML file and load images. The documentation could be constructed dynamically by gathering information from the XML file. The problem with this approach was that, it required a server to run server side scripts. One workaround would be to use a single server to host the documentation for all the people in an organization. However, a common server requires each update to be sent to the server. Another alternative was to allow users to modify the documentation once it is exported. Although this is convenient, it introduces another problem; the users might update the documentation but forget to update the model with the changes. One solution to this problem would have been to write another interpreter which would update the model with the changes. This would work if the users run the interpreter after they make changes to the documentation. The users are likely to forget to interpret after making changes to the documentation. This can result in inconsistencies. This approach would also require a server because changes cannot be made on the client side using just JavaScript and HTML.

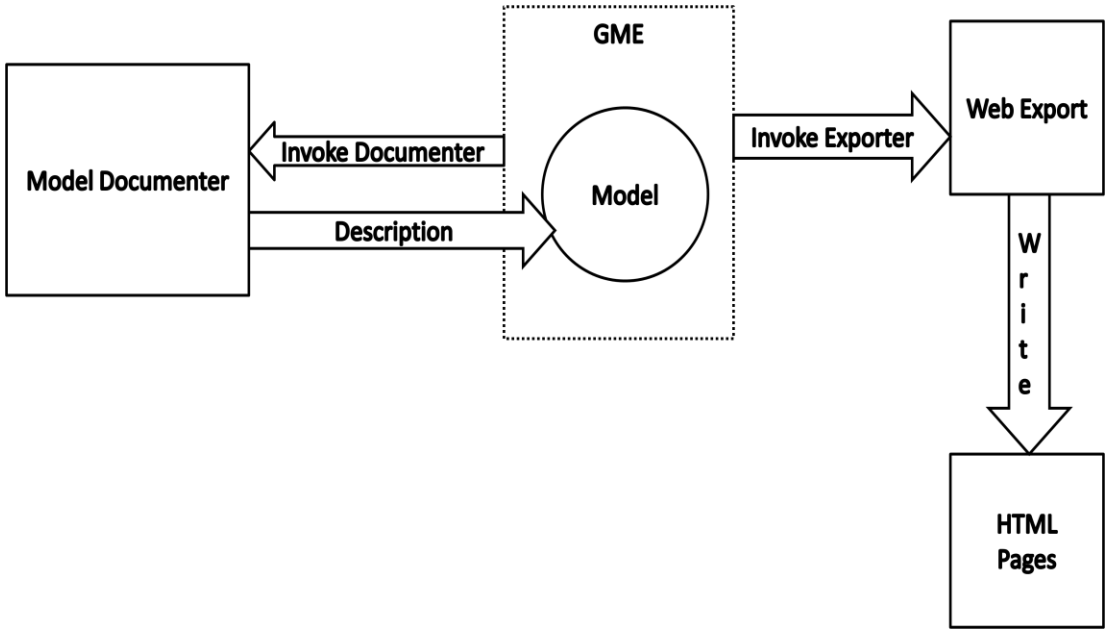

Figure 1 General System Architecture

Given the demerits of having a server, we eliminated all the design alternatives that required it. Our goal was to implement the system using only HTML and JavaScript. To avoid a server, all the necessary information is exported in the HTML files and is manipulated using JavaScript. As discussed in introduction, the system consists of two parts. The first part is the documenter which is used for entering descriptions. It is named “Model Documenter”. The other part is responsible for exporting the model into HTML Pages and is named “Web Exporter”. Model Documenter is used for writing descriptions about an object whereas the Web Exporter is used to export the graphical models, attributes and descriptions.

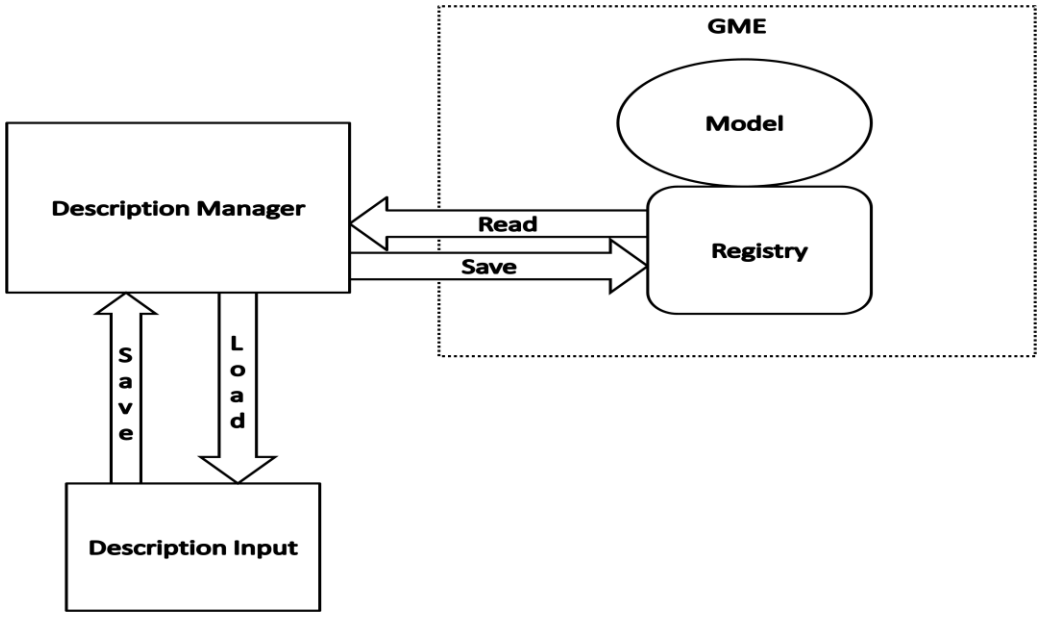

Figure 2 Model Documenter Architecture

Figure 1 shows the high level architecture of our system. The two components sit on top of GME and interact with GME to accomplish their tasks. Model Documenter can associate description with an object by interacting with GME. Web Exporter on the other hand obtains information from GME, organizes them in a HTML format and then writes to a file. Figure 2 and Figure 3 shows the architecture of the Model Documenter and the Web Exporter respectively. The Model Documenter is quite simple and has an input module which takes description from the user. The central component in the Model Documenter is the Description Manager which is responsible for managing descriptions. When the interpreter is invoked, the Description Manager is responsible for loading the descriptions into the Input module. The users can then add, modify or remove descriptions. Once the user has finished editing the description, the manager pulls the data and sends it to the GME registry for

storage so that it can be retrieved later. The Web Exporter also has a central component Documentation Creator which generates documentation with the help of other components. It uses PNG Converter to convert EMF images into PNG format. HTML tags are created using HTML tag writer and then are written to an HTML file. The location and the size of the objects in the GME are obtained by using Location Reader component. The Tree Generator generates hierarchical representation of the objects that can be navigated and the HTML File writer is used to write the HTML files.

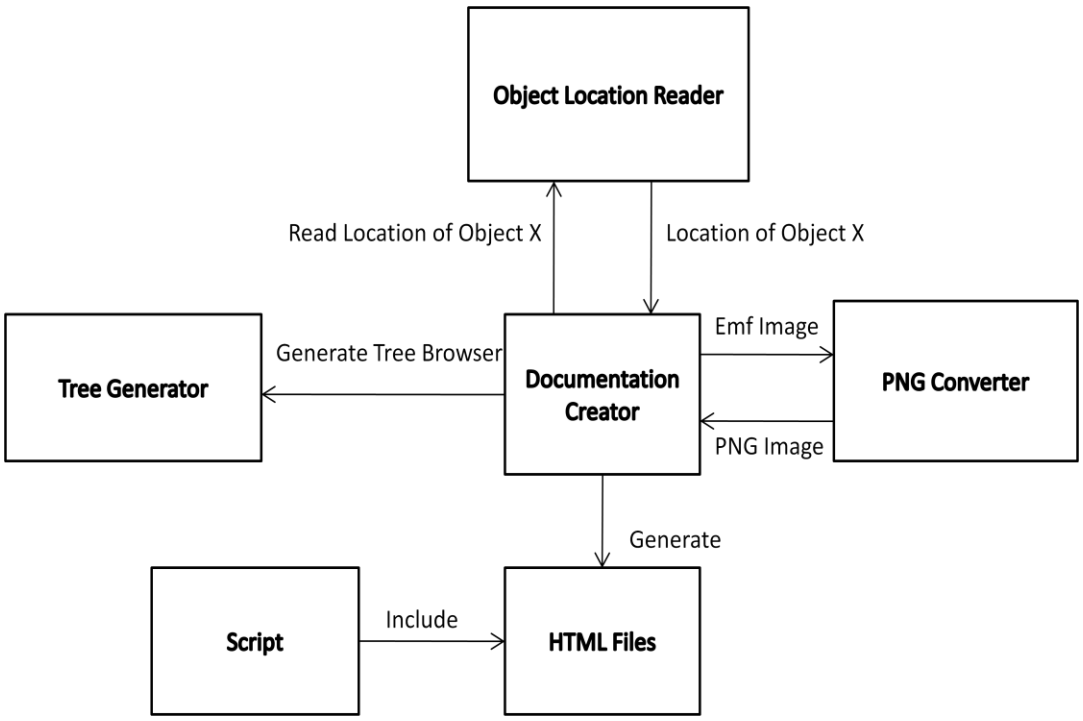

Figure 3 Web Exporter Architecture

CHAPTER IV

IMPLEMENTATION

Interpreter

GME provides programmatic access to a model. One popular method for access is to write an interpreter. Interpreters can be useful for manipulating models, generating configuration files, generating program code, or transforming models from one format to another to ensure compatibility [10]. Interpreters are generally written in C++ and are accompanied by classes and utilities that make the job of writing it really easy. It also has support for Visitor Pattern which can really be useful for traversing the model hierarchy.

HTML/ JavaScript

HTML is more attractive than plain text because it supports formatting, inclusion of links, images etc. It doesn't need anything extra than a web browser. JavaScript on the other hand helps add interactivity to the HTML pages. Using only JavaScript and HTML the need for a server is eliminated and the user has a freedom in generating and using the documentation.

The system is implemented as two interpreters and consists of the Model Documenter and the Web Exporter. The Model Documenter is used for describing an object. The description can be entered in HTML format. Therefore, it can be formatted as desired. The description entered is saved in the GME registry and is loaded when the Model Documenter is invoked for the object. The Web Exporter navigates each and every element in the model and exports its graphical view, attributes and descriptions into HTML pages. The generated

pages are interactive and allow navigation similar to the GME. The navigation is implemented using JavaScript. The web page has four sections: tree browser, graphical view, attributes and description.

The implementation consists of various modules and is described in detail below.

Model Documenter

When it is invoked, it displays a window with two views: Source view and Design view. In Source View, there is an input textbox for each aspect that the element is visible in (figure 4). There is an additional tab that takes the description for all aspects (the documentation will apply to all aspects). The textbox takes plain text as input which means that all the HTML tags needed for formatting must be coded manually. In Design View, the user can directly insert description in the HTML format. The user can use keyboard shortcuts for appropriate formatting. The Design View component employed is the Internet Explorer (IE) HTML control. Therefore, it supports any shortcuts that are supported by the IE control. The program is in no way meant to have full fledged HTML designing and editing capabilities. Therefore, there is a support for using an external editor. The users can use their favorite editor to edit HTML. The changes made through external editor will automatically be taken by the program and saved in the registry. Details on using an external editor are explained later.

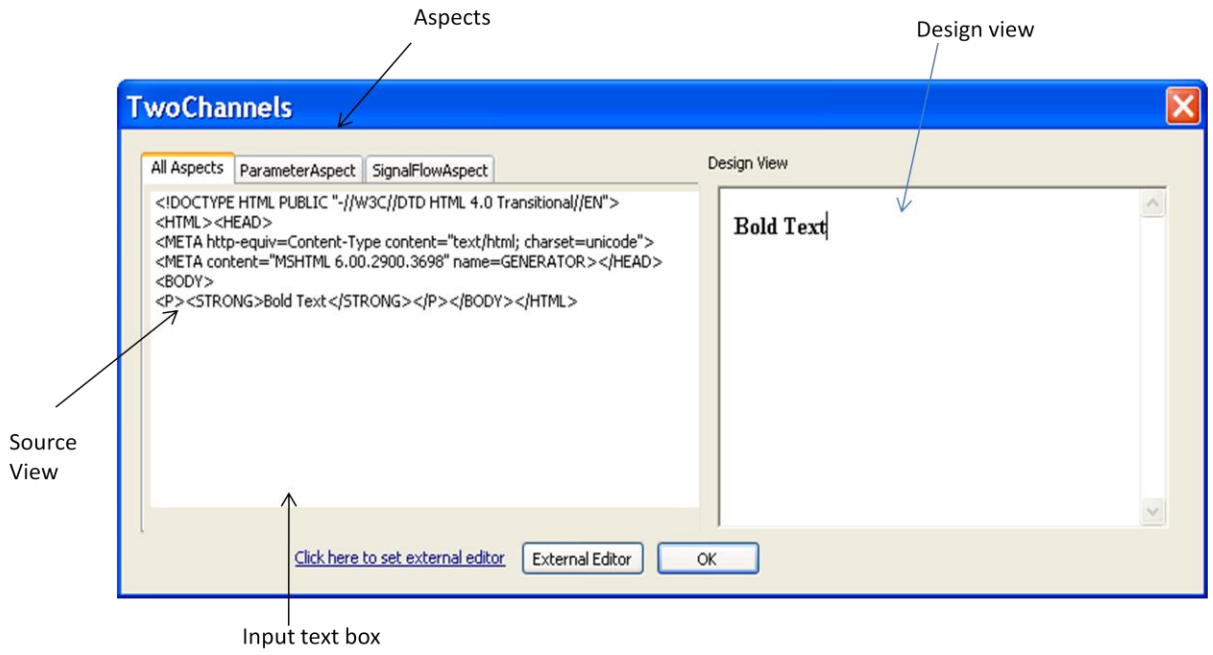

Figure 4 Model Documenter Window

Advantages of using HTML

Having an ability to insert documentation in HTML is always a plus. Users can always highlight the points they want to emphasize by using appropriate formatting (bold, italic etc.) Moreover HTML can be visually appealing to the readers of the documentation. Whenever one requires pointing to external material for explanation or wants to include a picture, it can be achieved using HTML.

Recording Documentation/Comments

Users can start the Documenter by selecting the object that they are interested in and then invoking the interpreter. A dialog box is displayed, asking the user for a description that he would like to associate with the object. The user can enter description for different aspects of the same object and also provide a description that is common to all aspects.

This is useful because the same description need not be copied to multiple aspects. Users can switch between the aspects and add, modify or remove text. Once the description has been edited, it can be saved and the user can restart working on the model. When the user saves the documentation, the interpreter goes through each aspect and collects the descriptions. Then it creates a description entry for each aspect in the GME registry. The entry is named using the identifier of the object followed by the aspect name. The description is saved under this entry. While retrieving the description, the interpreter first finds out the object identifier and the aspects that it supports. Then it constructs the registry entry name as mentioned before. If description is present, it is loaded in the appropriate textboxes. For example if an object has identifier *XYZ* and supports aspects *Aspect1* and *Aspect2*, then the first call to save documentation will create a registry node called *Description* and under it, it will create nodes for each aspect and an extra node representing all the aspects. Each description is stored under the respective aspect. Figure 5 shows how it looks like when exported in XML format. The figure doesn't show HTML tags for simplicity. The tags such as <a href>, <b> etc present in the model will automatically converted to XML acceptable format. Users can use hyper links to support their description. The link can be used by entering <a href ..> tag in source view.

```
<model id = “XYZ” >
  <regnode name = “Description” >
    <regnode name = “XYZ” >
      <value> …… </value>
        <regnode name = “Aspect1” >
          <value> …… </value>
        </regnode>
        <regnode name = “Aspect2” >
          <value> …… </value>
        </regnode>
    </regnode>
  </regnode>
</model>
```

Figure 5 Example of saving documentation in registry

External Editor

The users can use an external editor for HTML editing. Rather than making an attempt to provide a good HTML Editing feature, we thought it would be more beneficial for the users to be able to leverage the power of existing HTML editors. The user can register an external editor by clicking on the link provided at the bottom of the description input dialog box. The registered external editor’s path is saved in the Windows registry. The path could have been saved in the GME registry but the model may be copied across systems. The path of the external editor is dependent on the system and may not work when copied to another machine, as executable locations might differ. The path is saved under `HKEY_CURRENT_USER\Software\GME\Descriptions` with the key name *editor.* If the model is copied to another machine, the user can always register an editor that is present in the new machine. While loading the Model Documenter the Windows registry key is read and if it exists the editor is set. The user can launch the external editor anytime by clicking

on the *External Editor* Button. When the button is clicked, the system writes the current description into a file in the temporary directory and creates the external editor process by making the *CreateProcess* API call with editor and the temporary file as arguments. This will open the external editor with the description. The user can then edit and save the description. When the external editor is open, the GME window cannot take the user events, until the Model Documenter is notified that the editing is complete. This is achieved by displaying a message box that needs an acknowledgement from the user. As a result of this, the users are also prevented from switching to another aspect while the external editor is open. . If the users need to edit descriptions for two or more aspects, they should do it one aspect at a time. The message is acknowledged by pressing the OK button in the message box. After the acknowledgement is received, the Model Documenter loads the changes.

Syncing Source and Design View

As the user can edit both the source and the design, it's necessary for the system to synchronize the update in one view with the other. When the user is typing in the source view, the design view is updated only when the user stops typing for a second. This is done by recording the time stamp of the last edit and comparing it with the current time. There is a timer running every second which checks if the textbox has been inactive for a second. If it has been inactive for a second, the design view is updated. If user edits the textbox before a second expires then the design view is not updated. In the case of design view, there is no check for inactivity; the source view gets automatically updated every second. The source or design view is not updated if the design or source view hasn't been changed respectively.

Web Exporter

The Web exporter is used to export the model and the descriptions that are attached to the objects. The model is exported in the form of HTML pages and can be navigated. The steps involved in the export are:

1. Ask the user for a destination folder. This folder will be used for storing the output files. Attributes and descriptions are stored in subfolders that are created by the program.

2. Go through each aspect of each model in the hierarchy and export its view in the form of a Windows Metafile. Also export the model geometry in an xml file. The xml file contains the location of each object and its name. The location is present for every aspect of an object.

3. For each exported image, convert it into PNG format.

4. For all such images create an Image tag and add an Image map. The map will define a region for each object that is present in the image. Create Click and Double click JavaScript events for each of those regions.

5. Write attributes and descriptions of each object in the subfolder.

6. Write the HTML files.

The important processes in exporting to html pages are explained below:

Exporting model image

In order to export the model into web pages, the first step is to export the graphical view of the model. GME allows models to be exported as Enhanced Metafile (EMF). GME exposes this functionality as a COM interface and thus, the interpreter calls the interface to export the graphical view of the model. EMF is an image file format developed by Microsoft and is used in Windows based systems.

Conversion to PNG

Since EMF is a Windows based and relatively uncommon format, we opted for a format that is compatible across different operating systems. This led PNG (Portable Network Graphics) to be chosen as the image format. PNG is a lossless format and is supported across many browsers.

In the implementation, the presence of PNG encoders across different Windows based systems has been exploited. Using the encoder that is built into GDI+, a Windows API that helps application programmers write graphical applications, the EMF formatted image is converted to PNG format. The conversion is achieved in two steps:

1. First step is to obtain the encoder; this is done by enumerating all image encoders that are available in the system and searching for the PNG encoder. The encoders can be obtained by making *GdiPlus::GetImageEncoders()* call. Once the list of encoders is obtained, the PNG encoder can be searched. The search matches the PNG format name with the format name of each encoder in the list.

2. After obtaining the encoder, a new *GdiPlus::Image* object from the EMF file is created. The image object is then saved to a new file. The save method of the image object takes an image encoder as a parameter. This leads the format conversion from EMF to PNG.

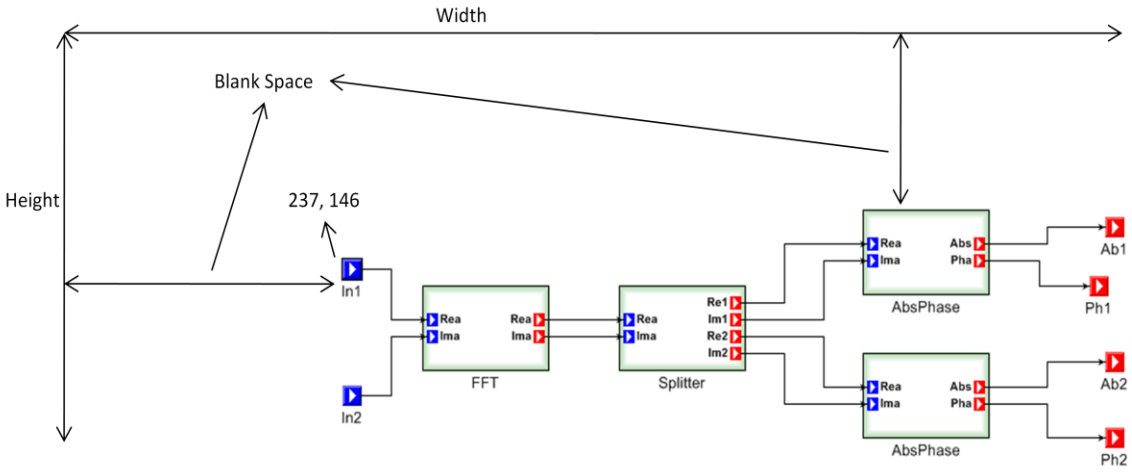

Figure 6 Original model as it appears in GME

While exporting EMF files, the GME truncates extra spaces that surround the model. This is the default behavior of the Windows API that is used by the GME to export a model. The truncation is useful as it makes the image look neat and professional. But it introduces a problem of mapping the location in the image that corresponds to the location in the GME window. The problem is illustrated in figure 6. The figure shows the original model in the GME window. The figure has extra spaces at the top and the left. The location reported by the GME is the location in the GME window. For example, in Figure 5 the leftmost and topmost object coordinates was (237,146). The same view exported to an image (as shown

in figure 7) has the coordinates of (3, 46). The problem is solved by mapping the smallest x and y coordinates reported by the GME to the smallest x and y coordinates found by searching the image. The details of the solution are discussed later.

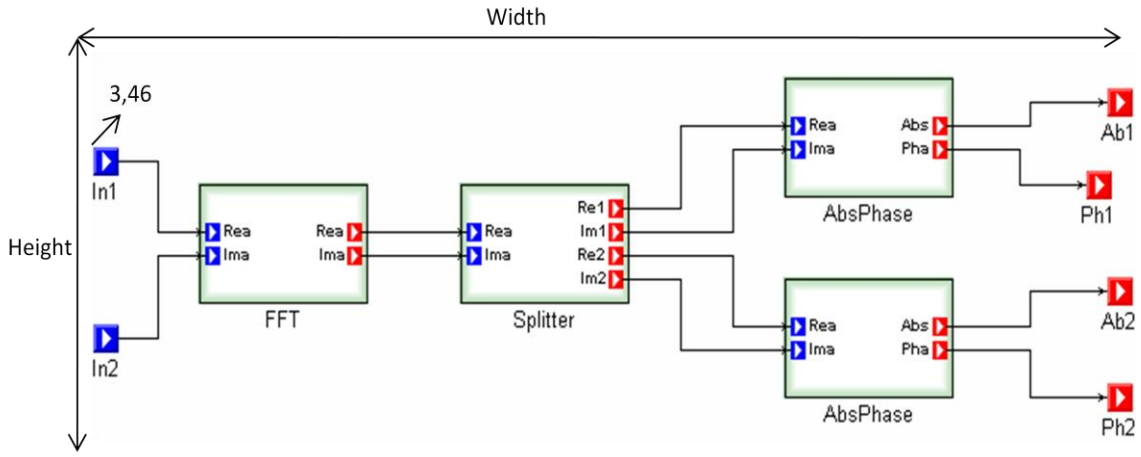

Figure 7 Exported model from GME

Exporting Model Locations

The GME registry only provides the top left corner coordinate of an object. The location and bounds can be obtained by calling the GME interface that exports object locations and boundaries in a XML file. The exported file contains locations of every object contained in the model. The locations are present in the file for every aspect that an object is visible in. The interpreter reads the object location from the file. The XML files are exported for every model present in project. This is an inconvenient approach and suffers from overhead of

reading XML files. It would have been better if the GME interface had provided a proper API for reading an object's bounds.

Mapping location

As discussed earlier, the truncation of exported image necessitates mapping the location obtained from the XML file to the location in the image. The problem is solved by finding the minimum x and y coordinates in the xml file and searching the exported image for the minimum x and y coordinates. The two coordinates are equivalent. Based on these two coordinates, a correction can be applied to obtain an object's position in the exported image. The algorithm is given below:

1. MinX = MinY = infinity
2. Search for minimum x1 and y1 coordinate in XML file (x and y can be coordinates of different objects)
3. Start reading value of pixels from top left corner moving towards right until x = MinX or x = Image Width
4. If current pixel (x2,y2) is not the background color and x2 < MinX then MinX = x2
5. Similarly obtain MinY
6. For any object location (x,y) apply correction x = x + x1 – MinX and y = y+ y1 – Miny

This algorithm is based on the fact that the GME exports the images without including background color. This algorithm takes O (mn) time where m = vertical resolution and n = horizontal resolution of the image. Though worst number of searches is m * n, it is always less than that because minimum x and y values will be found before all the pixels are searched. To make this efficient, we could only search the rectangle formed by top left hand corner and the minimum x and y coordinates obtained from the xml file. However, it was observed that sometimes space gets added to the image. This is the case when the image has very little extra space before or after the objects in the model. So using minimum coordinate from the XML file, may never find the minimum coordinates in exported image. We can get around this by searching the whole image.

Exporting to HTML

The exported image is divided into regions using an image map. Each region represents an object in the image. The regions are obtained by mapping locations as described earlier. Each area in the image map is associated with JavaScript *onClick()* and *onDblClick()* events. The necessary information needed for navigation is written while generating the HTML files. The *onClick()* event handler currently takes 5 parameters. Those parameters are coordinates of the object, its unique ID assigned by the GME, and 3 Boolean values indicating whether it has children, attributes and documentation. For example, if an object has attributes then it will be set to true in the event handler call. onDblClick() event handler also takes same set of parameters excluding the coordinates.

ID is needed as a parameter because the attributes, descriptions and html files are named using the ID. This makes it easy to find the appropriate file. A Boolean value, which indicates whether the object has children, attributes and description respectively, is used to

decide whether such HTML documents exist. It's needed because JavaScript and *iframe* have no explicit way of knowing whether the given file exists. This is particularly true for web browser environments because of the security reasons.

Initially, all the aspects that were used by the paradigm were listed at the top of the view. But it is desirable to display only those aspects that are supported by the current model. Therefore, we decided to display only those aspects. The aspect names that appear at the top of the window, are in a separate *iframe*. Whenever the view is changed, the aspect supported by the current view is loaded. To achieve this, the aspects that are supported by the model are placed in a hidden *div* tag and when the HTML page is loaded the data from hidden *div* tag is copied to the aspect *iframe*. So only the aspects that the object supports are available.

Generating the Tree Browser

The Tree Browser component is taken from the Yahoo UI Library. The library requires the tree hierarchy to be represented by <ul> and <li> tags. The interpreter navigates the object hierarchy and puts sibling objects in the same list and the immediate descendants in the child list to fulfill the Tree Browser Component requirement. Figure 8, shows a simple example of how the Tree Browser is constructed. Additional information (not shown in figure) is also present in the list items. The information include the ID, the ID of the parent, type of the object, aspects that the object has and the Boolean values indicating whether the object has child, attributes and documentation. Except for two new arguments, others have similar meanings as discussed earlier in Exporting to HTML. For example, the aspect name and ID are used to construct the file name of the appropriate model that is to be loaded in the *iframe*. The two new arguments are Parent ID and Aspects. Parent ID was needed

because when an item is clicked on the Tree Browser, it is loaded in the view *iframe*. However, the leaf objects (atoms, sets etc) don't have views. So it was appropriate to load the parent view for such objects. To locate the parent view, Parent ID is required. When an object is opened in the parent view, the child object is highlighted. If the system is currently in a different aspect that is not supported by the object then a random aspect is selected. The selected aspect is used to display the object.

Object Hierarchy	**HTML Code Using List**
RootFolder ModelA AtomA ModelB AtomB	 - RootFolder - Model A AtomA - ModelB - AtomB

Figure 8 Object Hierarchy and corresponding HTML code which is read by Yahoo library to construct tree browser

Navigation

The exported model is navigated using JavaScript. Clicking an object in the view selects the object and double clicking an object brings the object into view. The selection of object is achieved by making the object bordered with thick black rectangle. Internally, this employs a

div whose border is set to thick black using CSS. When user clicks on the object, the *div*'s coordinates are made equal to the object's coordinates. This makes a *div* layer to appear above the object. This creates a problem as the click and double click now go to the *div* layer. To circumvent this problem, the *onClick* and *onDblClick* event handler of the image area is set as the event handler for corresponding events of the *div* layer.

Whenever an object is clicked, the aspect name and ID are used to construct the attribute and description filename. The filename is the Aspect name followed by an underscore which is followed by the object ID. Thus *onClick()* checks whether object has attributes and description using the Boolean parameters. If the parameters are true, the file is loaded in the respective *iframe*. If no such files exist, then 'No attributes' or 'No Description' message is displayed.

CHAPTER V

CASE STUDY

Signal Flow Example

The case study applies the interpreters to the Signal Flow example model that comes with the GME. The first step before exporting the model is to describe objects in the model. As discussed earlier, this can be done by running the Model Documenter interpreter. A particular object in the GME window is selected and then the interpreter is invoked. This will

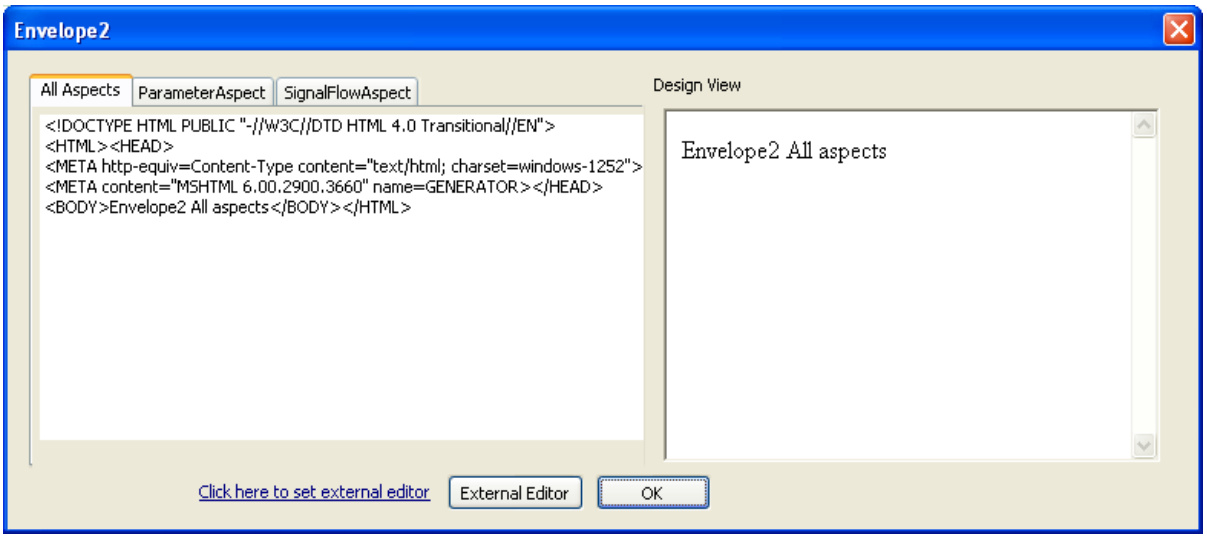

Figure 9 Model Documenter window that takes documentation

pop up a dialog as shown in Figure 9 (in this case the interpreter has been run for the Envelope2 object in the model). Tabs are created for every aspect that the object supports and the user can switch between tabs to enter a description. The source view is on the left and the design view is on the right. The source view takes plain text as input. So any

required HTML tags have to be written manually. The design view on the other hand can directly accept HTML tags. For example, typing a bold text in source view requires specifying and tags whereas in design view it is a simple matter of pressing Ctrl + B shortcut key. The editing capability is pretty basic and therefore use of an external editor is supported. The external editor can be launched by pressing External Editor Button. Once the external editor is launched, description can be written using the editor. After editing is finished, the editor needs to be closed. The changes would then be reflected in the Model Documenter window. The description is saved when the user clicks the OK button.

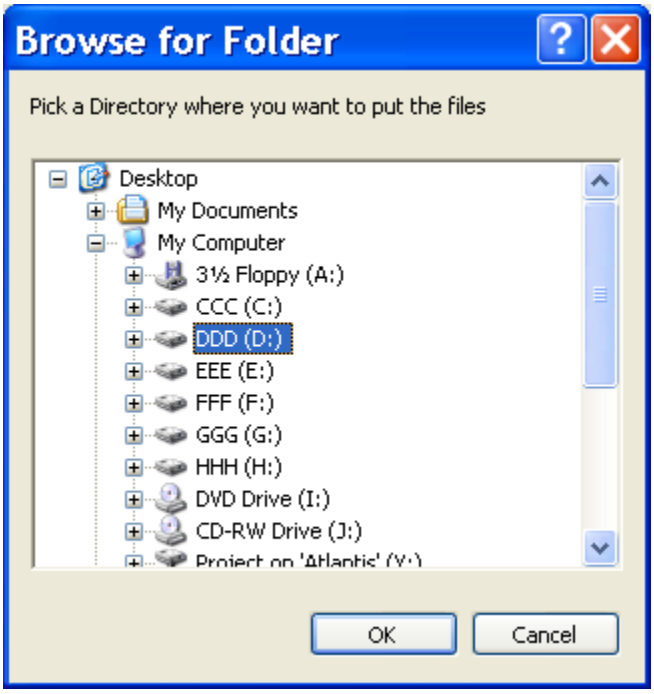

Figure 10 Dialog asking to select a folder

The Web Exporter tool is launched to export the model into HTML pages. The tool asks the user to select a destination folder where the files should be stored. The dialog is shown in figure 10. The folder can be selected by navigating the tree browser. Once the

proper folder is selected, the OK button is pressed to designate the folder as the export folder. The tool then starts exporting the files. This will take some time because it requires processing each and every object present in the model. When the export is complete, it displays a message saying the files have been exported successfully. When the message is acknowledged by pressing the OK button, default web browser is launched with the *tree.html* file. The model can then be browsed in the web browser.

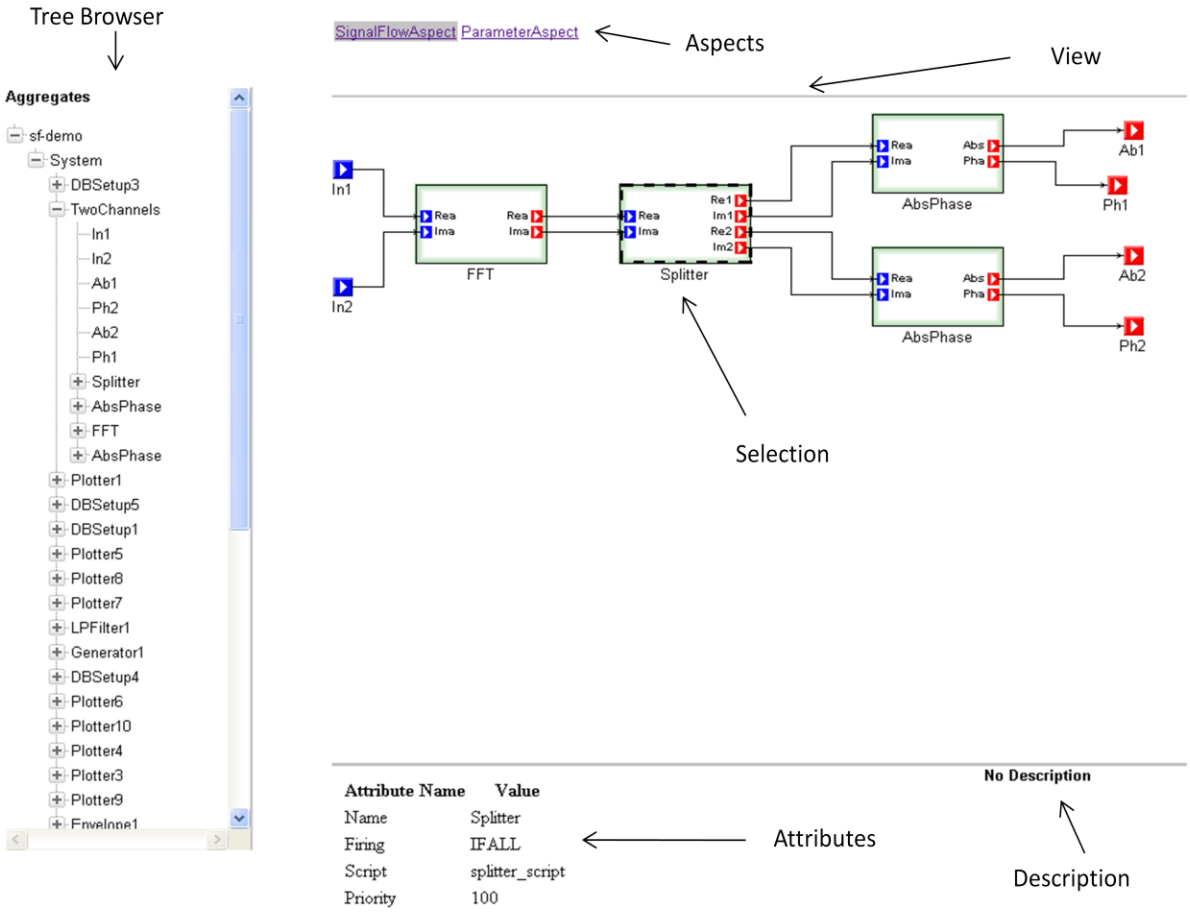

Figure 11 Exported HTML in Web Browser

Figure 11 shows how the exported document looks like. As said earlier, there is a tree browser on the left. This browser contains all the objects in the hierarchy and is expandable. At the top is always the root element followed by its descendents. If a model has child objects then they will be the child nodes of the model node. Clicking an item in the browser will bring the item into view. For example if a model is clicked, it will bring that model into view. If the model has children then the model's view is opened, else its parent view is opened. This is not true for folders because they don't have views.

On the top of window there is an Aspects tab which allows users to switch between aspects. Only the aspects that are applicable to the current model in view is displayed. Below the Aspects tabs, there is a frame which shows the model. The objects on the model can be selected by clicking on them. Clicking the object displays its attributes on the bottom left hand side and the description on the bottom right hand side. One can also double click the object to bring it in the view. This is true for models that have children. For atoms and sets, it does nothing. For Reference types, double clicking will load the referred object's parent with the referred object highlighted.

CHAPTER VI

SUMMARY AND FUTURE WORK

In this thesis, we built the Model Documenter and the Web Exporter. The tools run as GME interpreters and can be invoked from the GME window. The documenter is used to associate description with the objects in the model. Using the documenter, the user can explain the model, which can be helpful to other users. External references, links etc. can be used for the explanation. The exporter is used to export graphical view of the models, attributes and descriptions to HTML pages. The exported pages can be navigated to view the model, attributes and description. The HTML pages can be viewed by any user with a web browser. This makes it easy to share models among users.

The tools can be improved in different ways. The model export tool lacks support for Connections since they are not currently selectable in the exported model and their attributes and descriptions are not available. Connections are difficult to handle because their shapes can consists of several line segments. This could be solved by using single *div* tag for each line segment, but this can be complicated when connections cross each other. Currently, attributes and descriptions are only visible for objects other than Folders. The system can be extended to support folders. Another improvement for the model export tool would be to use the SVG format instead of the PNG format. SVG format stores images as a set of points, lines and other shapes and therefore, the quality of the image is maintained when scaled up.

REFERENCES

[1] Miliaev, N., Cawsey, A., and Michaelson, G. 2002. Technical Documentation: An Integrated Architecture for Supporting the Author in Generation and Resource Editing. In *Proceedings of the 10th international Conference on Artificial intelligence: Methodology, Systems, and Applications* (September 04 - 06, 2002). D. R. Scott, Ed. Lecture Notes In Computer Science, vol. 2443. Springer-Verlag, London, 122-131.

[2] Harold Thimbleby, Combining Systems and Manuals. In Proceedings Conference on Human-Computer Interaction, HCI'93, Vol. VIII, BCS.

[3] Reiter, E., Mellish, C., and Levine, J. 1998. Automatic generation of technical documentation. In *Readings in intelligent User interfaces*, M. T. Maybury and W. Wahlster, Eds. Morgan Kaufmann Publishers, San Francisco, CA, 141-156.

[4] http://java.sun.com/j2se/javadoc/

[5] http://www.mathworks.com/

[6] http://www.microsoft.com/visio

[7] http://www.ibm.com/software/rational/

[8] http://jude.change-vision.com

[9] Sztipanovits, J. and Karsai, G. 1997. Model-Integrated Computing. *Computer* 30, 4 (Apr. 1997), 110-111.

[10] Karsai, G., Ledeczi, A., Neema, S. and Sztipanovits, J. "The Model-Integrated Computing Toolsuite: Metaprogrammable Tools for Embedded Control System Design". Proc. of the IEEE Joint Conference CCA, ISIC and CACSD, Munich, Germany, 50-55, October, 2006.

[11] Ledeczi, A., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J., and Karsai, G. 2001. Composing Domain-Specific Design Environments. *Computer* 34, 11 (Nov. 2001), 44-51.

[12] Karsai, G.; Sztipanovits, J.; Ledeczi, A.; Bapty, T.; , "Model-integrated development of embedded software," *Proceedings of the IEEE* , vol.91, no.1, pp. 145- 164, Jan 2003.

[13] Ledeczi, A.; Balogh, G.; Molnar, Z.; Volgyesi, P.; Maroti, M.; , "Model Integrated Computing in the Large," *Aerospace Conference, 2005 IEEE* , vol., no., pp.1-8, 5-12 March 2005.

[14] Lurdes, J., and Carapuca, R. Automatic generation of documentation for information systems: Lecture Notes in Computer Science, vol. 593. Springer, Berlin, 48-64.

[15] Estevez, E.; Marcos, M.; Sarachaga, I.; Lopez, F.; Burgos, A.; Perez, F.; Orive, D.; , "Model based documentation of automation applications," *Industrial Informatics, 2009. INDIN 2009. 7th IEEE International Conference on* , vol., no., pp.768-774, 23-26 June 2009.

[16] Murphy, S. and MacKinnon, N. 2008. Designing UML and UML-based diagrams for technical documentation: where are we now?. In Proceedings of the 26th Annual ACM international Conference on Design of Communication (Lisbon, Portugal, September 22 - 24, 2008). SIGDOC '08. ACM, New York, NY, 9-14. DOI= http://doi.acm.org/10.1145/1456536.1456539.